A Framework for Automatically Generating Interactive Instructional Scaffolding

Eleanor O'Rourke[1], **Erik Andersen**[2], **Sumit Gulwani**[3], **Zoran Popović**[1]
[1]Center for Game Science, Computer Science & Engineering, University of Washington
[2]Department of Computer Science, Cornell University [3]Microsoft Research Redmond
{eorourke, zoran}@cs.washington.edu, eland@cs.cornell.edu, sumitg@microsoft.com

ABSTRACT
Interactive learning environments such as intelligent tutoring systems and software tutorials often teach procedures with step-by-step demonstrations. This instructional scaffolding is typically authored by hand, and little can be reused across problem domains. In this work, we present a framework for generating interactive tutorials from an algorithmic representation of the problem-solving thought process. Given a set of mappings between programming language constructs and user interface elements, we step through this algorithm line-by-line to trigger visual explanations of each step. This approach allows us to automatically generate tutorials for *any* example problem that can be solved with this algorithm. We describe two prototype implementations in the domains of K-12 mathematics and educational games, and present results from two user studies showing that educational technologists can author thought-process procedures and that generated tutorials can effectively teach a new procedure to students.

Author Keywords
Computational education; scaffolding; authoring tools.

ACM Classification Keywords
H.5.0. Information Interfaces and Presentation: General

INTRODUCTION
Procedural knowledge is required for a wide range of human activities, from cooking to mathematics to using software applications. Many of the concepts taught in interactive learning environments like cognitive tutors, educational games, and software tutorials are procedural in nature. Some procedures, such as opening a file in a text editor, are simple. Others, such as long division, involve complex control flow structures like loops and conditionals. Teaching complex procedures in interactive learning environments presents many challenges.

One effective method of teaching procedures is by providing scaffolding that helps students solve problems that would otherwise be beyond their reach [11, 27]. While there are many different approaches to scaffolding learning, interactive learning environments typically use step-by-step demonstrations, tailored progressions of practice problems, and individual feedback to support students. Intelligent tutoring systems provide student with adaptive sequences of problems and personalized hints [4, 10, 26], and software tutorials often use step-by-step demonstrations to teach procedural tasks [6, 14].

While interactive scaffolding is effective, this type of content is often time-consuming to create because it must be authored by hand [1, 12]. As a result, many researchers have explored methods of generating interactive explanations automatically [2, 6, 7, 9, 12, 13]. While these approaches have significantly reduced the amount of effort required to author scaffolding, they often apply to only one application domain or require the designer to write separate content for each example problem. In some domains, such as K-12 mathematics, we may want to demonstrate a procedure for a large number of practice problems and produce multiple different types of scaffolding. Ideally, we would like an application-independent method of explaining a given procedure for *any* example problem.

In this paper, we introduce a new framework for automatically generating interactive tutorials to teach the entire space of problems that can be solved with a given procedure. In our framework, an educational technologist encodes the problem-solving thought process as an algorithm and provides a mapping between programming language constructs and visual objects in the user interface. By stepping through this algorithm line-by-line, we automatically generate visual explanations for each step. This approach has a number of advantages. Given a set of mappings and an encoded algorithm, our framework automatically produces tutorials for any example problem in the domain. Furthermore, our framework naturally supports the creation of many different types of interactive scaffolding, including problem progressions, just-in-time feedback, and fading worked-out examples.

We demonstrate the generality of our approach by showing how it applies to two distinct problem-solving domains: a K-12 mathematics application and an educational game. Through a user study with fifth-grade students, we show that our generated tutorials can produce learning gains similar to those of a human teacher. We also show that that educational technologists can encode an algorithm to explain subtraction in just a few hours. We believe this approach can greatly improve the process of authoring interactive scaffolding, particularly in domains with large numbers of practice problems.

CHI 2015, April 18 - 23, 2015, Seoul, Republic of Korea
Copyright 2015 ACM 978-1-4503-3145-6/15/04$15.00
http://dx.doi.org/10.1145/2702123.2702580

RELATED WORK

Researchers have explored many methods of designing and authoring interactive instructional materials to teach procedural knowledge through computer applications. In this section, we provide an overview of existing work in two domains: intelligent tutoring systems and software tutorials.

Intelligent Tutoring Systems

Intelligent tutoring systems (ITS) are applications that emulate one-on-one tutoring with a human teacher [4, 10, 26]. These systems teach procedural concepts using personalized progressions of practice problems and contextual feedback [26], and they have been shown to produce learning gains in classroom studies [15]. While ITSs have great potential, they have not been widely adopted because they are expensive to build [10]. It has been estimated that 200-300 hours of expert design is needed to produce one hour of ITS content [1].

Researchers have explored many methods of reducing tutor authoring effort. Brawner et al. provide design recommendations for authoring tools, highlighting the importance of domain-independent methods [8]. Sottilare describes efforts to develop a Generalized Intelligent Framework for Tutors based around a standard ontology for representing knowledge [24]. Stamper et al. developed the Hint Factory to make it easier to add ITS-style interaction to existing educational software by using Markov decision processes to automatically generate contextual hints from past student data [25].

Others have focused on reducing the expertise required to author content for an intelligent tutoring system. Blessing et al. developed a cognitive model SDK that allows authors to edit rule hierarchies through a user interface to reduce the need for programming expertise [7]. Aleven et al. developed the Cognitive Tutor Authoring Tools (CTAT) to allow designers to create example-tracing tutors without programming [2]. With CTAT, designers demonstrate correct and incorrect procedures for an example problem to create a "behavior graph" that can be generalized to tutor multiple isomorphic problems. CTAT has been shown to reduce development costs by a factor of 4 to 8 [2]. Olsen et al. extended CTAT to support the authoring of collaborative ITSs built around collaborative scripts [18]. We take a different approach to improving the scaffolding authoring process. Rather than designing tools to reduce the expertise required to author tutors, we aim to maximize the content that programmers can create.

Software Tutorials

Since many tasks in software applications involve following procedures, there is a body of research in software usability that explores methods of teaching procedural knowledge. This work has shown that step-by-step tutorials are effective instructional tools, especially when they provide interactive elements such as stencils [14], videos that track the user's progress [20], and interface highlighting [6].

While step-by-step tutorials are effective, authoring this content can be time-consuming [6, 12, 21]. To address this issue, some researchers have focused on improving the quality of video tutorials, since these are comparatively low-cost to create. Pongnumkul et al. developed Pause-and-Play, a system that links events in a video to user events in the target application, pausing the tutorial when the user lags behind [20]. Others have focused on adapting existing tutorials for new domains. Ramesh et al. developed ShowMeHow, a system that automatically transfers step-by-step tutorials between similar applications such as Photoshop and GIMP [21].

Researchers have also explored methods for generating tutorials automatically [9, 12]. One approach is to generate tutorials directly from user demonstrations of a procedural task. Grabler et al. developed a system for automatically generating text and image tutorials from demonstrations in an instrumented version of GIMP [12]. Chi et al. developed MixT, which automatically generates mixed media tutorials with both text and video from user demonstrations in Photoshop [9]. While these systems produce tutorials automatically, they only work for specific domains like image manipulation, and the tutorials cannot always be applied to new problems. Furthermore, these systems only support linear procedures. In contrast, our framework supports complex non-linear procedures, applies to multiple domains, and can generate tutorials for any example problem that can be solved by the procedure.

Another approach is to generate tutorials that show users how to recreate an artifact in a user application. For example, Harms et al. augment a programming environment to automatically generate stencil-based instructions from code snippets [13], and Li et al. automatically generate tutorials to recreate AutoCAD drawings [16]. These systems can automatically produce tutorials for multiple example problems, but provide little control over how the tutorial teaches the procedure. In educational domains, it is important to teach students by describing the problem-solving thought process.

The work that relates most closely to ours is the DocWizards system by Bergman et al., which learns a procedure from multiple user demonstrations [6]. The DocWizards interface exposes a human-readable version of the learned procedure so a new user can step through it line-by-line. At each step, the current line is highlighted and the widget involved in the next action is circled in the user interface. While the DocWizards system uses an algorithmic representation of a procedure to automatically generate tutorials, it was not designed to explain the thought process. The learner is expected to read and understand $if - else$ statements, which significantly reduces the usability of this system. Furthermore, the procedure must be demonstrated on many distinct problems to create an accurate underlying algorithm, and the complexity of the procedures that can be represented is limited.

FRAMEWORK DESIGN

We present a new framework for generating step-by-step tutorials to teach procedural knowledge. In our framework, an educational technologist encodes the problem-solving thought process as an algorithm. Using a domain-independent interpreter, we step through this algorithm line-by-line. We explain each line using a set of mappings between programming language constructs and visual objects in the user interface, provided by the educational technologist. This approach allows us to automatically explain any problem that can be solved with the procedure. Figure 1 shows this workflow.

Algorithm 1 Given as input a 2D array $table$ containing a minuend p, represented as a set of digits $p_0, p_1, ..., p_m$ located in cells (0,0) to (0, m), and a subtrahend q, represented as a set of digits $q_0, q_1, ..., q_n$ in cells (1,$m-n$) to (1, m), subtract:

```
1: procedure SUBTRACT(table)
2:     for each column in table.GetColsFromRight() do
3:         top := column.GetCell(0)
4:         bottom := column.GetCell(1)
5:         if bottom.NotEmpty() then
6:             if top.value < bottom.value then
7:                 borrow := table.LeftOf(top)
8:                 while borrow.value == 0 do
9:                     borrow := table.LeftOf(borrow)
10:                end while
11:                borrow.value := borrow.value − 1
12:                while borrow.NotNextTo(top) do
13:                    borrow := table.RightOf(borrow)
14:                    borrow.value := 9
15:                end while
16:                top.value := top.value + 10
17:            end if
18:            result := column.GetCell(2)
19:            result.value := top.value − bottom.value
20:        else
21:            result := column.GetCell(2)
22:            result.value := top.value
23:        end if
24:    end for
25: end procedure
```

Thought Process Language

To complete a procedural task, a student must follow a certain thought process to decide when to perform each action. A simple procedure may be a list of actions to be executed in order, while a complex procedure will require the student to choose which steps to execute based on features of the current problem. To solve a subtraction problem, for example, the student needs to decide whether or not to borrow by comparing the top and bottom digits in a column.

We encode this problem-solving thought process in a language we call the Thought Process Language (TPL). The TPL algorithm for subtraction is given in Algorithm 1. This algorithm solves all single-digit and multi-digit subtraction problems. TPL constructs are standard to many programming languages, and are not a contribution of this paper. TPL includes basic primitives like variables and constants, as well as expressions that can include integer operators ($+$, $-$, $*$, $/$) or Boolean operators ($<$, $>$, $==$, $!=$). TPL has three types of statements: assignments, conditionals, and loops (for, while, do-while, for-each). TPL is based on object-oriented programming languages and also supports objects, object fields, and object methods. Additionally, TPL includes functions and a procedure can reference multiple sub-procedures.

Compiler and Interpreter

We generate explanations for a given problem by stepping through the TPL algorithm line-by-line like a debugger. To accomplish this, we use a compiler and an interpreter that are designed to be reused across applications. The TPL algorithm is compiled and passed into the interpreter, which maintains information about the set of variables that are currently stored in memory. As the interpreter executes each line, it broadcasts events about this state, such as when new variables are put into context, when statements are executed, and when sub-procedures return. A full list of the events thrown by the interpreter is given in Table 1. Each event contains relevant information like the name of the variable, the programming language object associated with the executed statement, or the name of the sub-procedure that was called. These events are passed to domain-specific interface hook functions, which generate and display visual explanations.

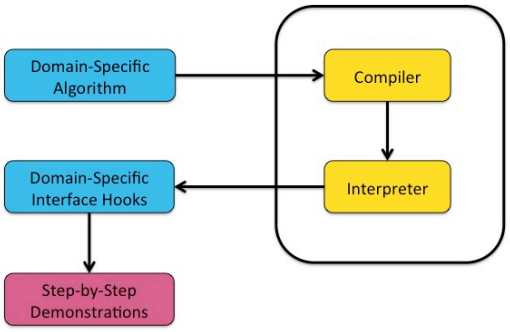


Figure 1. The framework workflow. A domain-specific TPL algorithm representing the problem-solving thought process is given as input, compiled, and then passed to the interpreter. The interpreter steps through the algorithm line-by-line and throws events, which are passed to the domain-specific interface hooks, which generate visual explanations.

Event	Interface Hook
Put variable in context	Interface Highlighting
Remove variable from context	Interface Highlighting
Execute assignment statement	Textual Explanation
Execute conditional statement	Textual Explanation
Execute loop statement	Textual Explanation
Execute return statement	Textual Explanation

Table 1. The type of interface hook used to explain each type of event.

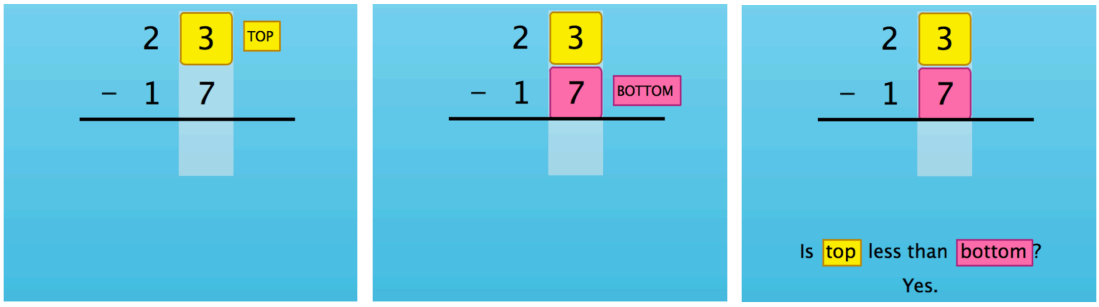
Interface Hooks

To explain the events that are broadcast by the interpreter, we use domain-specific functions called *interface hooks* that map events to visual explanations in the user interface. We use two types of visual explanations: textual descriptions and interface highlighting. The explanation used for each type of interpreter event is shown in Table 1. Authoring interface hook functions involves writing code that is similar to what a developer would write to create problem-specific tutorials for an application, but our framework allows this code to be re-used to explain any problem in the given domain.

Textual Explanations

Textual explanations are commonly used to teach procedural tasks in both software tutorials and intelligent tutoring systems [9, 26]. We use text to describe assignment statements, return statements, loops, and conditionals. To display a textual explanation in our framework, we first automatically generate the explanation and then display it in the user interface.

We generate explanation text in a completely application-independent manner using information that is encoded within the TPL algorithm itself. We use templates to convert lines of TPL code into textual explanations. A template is a single-sentence string that describes a certain type of statement, for example a conditional statement. To generate the explanation for a line of code, we fill the appropriate template with the



(a) (b) (c)

Figure 2. Examples of the explanations generated for the subtraction procedure. Figure (a) shows the explanation of the assignment of a variable called *top* to the top cell in a column. Figure (b) shows the explanation of the assignment of a variable called *bottom* to the bottom cell. Since the variable *top* is still in scope it remains highlighted. Figure (c) shows the explanation of the condition statement that determines whether borrowing is needed.

names of the variables, properties, or functions that are used in that line. The template library includes 20 explanation templates, which are designed to be reused across applications.

Figure 2 shows explanations generated for the subtraction procedure in Algorithm 1. Assignment statements are explained using the name of the assigned variable, so the code $top := column.GetCell(0)$ in line 3 is explained with the text "Top". To explain the conditional statement $if\ top.value < bottom.value$ in line 6, we fill in the conditional template "<explain boolean expression> <explain boolean result>". We populate the boolean expression template "Is <variable one> less than <variable two>?" to get "Is top less than bottom? <boolean result>." Finally, we fill the boolean result template with either "Yes" or "No" based on whether the expression evaluates to true or false. This produce the final explanation "Is top less than bottom? Yes."

To display the generated textual explanations in the user interface, our framework relies on application-specific interface hook functions. One interface hook function must be written for each location in the user interface where the designer wants to display text. As a result, the number of interface hooks required is highly dependent on the complexity of the domain. For the subtraction example, text is displayed in two locations. One interface hook function displays text next to a grid cell when a variable is assigned to a cell, and another displays text below the problem for all other types of programming language statements.

Interface Highlighting

We provide context to our textual explanations by highlighting visual objects in the user interface, an approach that is used in many effective software tutorials [6, 14]. Our goal is to highlight the set of interface objects that are relevant to the current steps in the procedure, without overwhelming the student by highlighting too many objects at once. To accomplish this, we determine highlighting based on the scope of variables within the procedure context.

Most imperative languages have a sense of *variable scope*, or the context in which a variable is defined and can be referenced. A variable that is defined within a branch of a conditional statement, like the $borrow$ variable in the subtraction procedure, will only be in scope while that branch is executing. We draw a connection between variable scope and working memory, declaring that the objects a student needs to keep in working memory while executing a procedure are equivalent to the variables that are currently in scope. We use the scope of variables to determine whether the visual representations of those objects should be highlighted.

To highlight objects in the user interface, our framework relies on application-specific interface hook functions. One interface hook function must be written for each type of object defined as a variable in the TPL algorithm. In the subtraction example, we define four variables: top, $bottom$, $borrow$, and $result$. All four of the variables are grid cell objects, so in this example we use one interface hook function that highlights arbitrary grid cells in the user interface. Using this interface hook function, our system highlights the objects referenced by the top and $bottom$ variables while they are in context, as shown in Figure 2(b). Variables defined in the TPL algorithm must refer to visual objects in the user interface, however we have found that all variables do have visual representations in the procedures we implemented for our prototype systems.

Authoring Process

This framework utilizes domain-independent components where possible to reduce authoring effort. To apply this framework to a new application, an educational technologist first integrates the domain-independent compiler, interpreter, and explanation templates into their code base. Then, the developer writes interface hook functions that display textual explanations and highly visual objects in the user interface. Finally, the developer writes algorithms in TPL that encode the problem-solving thought process for each procedure to be taught. Note that the interface hook functions are only written once for a given application, and can then be used to explain multiple TPL algorithms. For example, in our grid mathematics prototype, one set of interface hooks can be used to explain a broad range of K-12 math procedures such as subtraction, long division, and fraction multiplication. The number of interface hooks required for a given application will depend highly on the complexity of the problem domain.

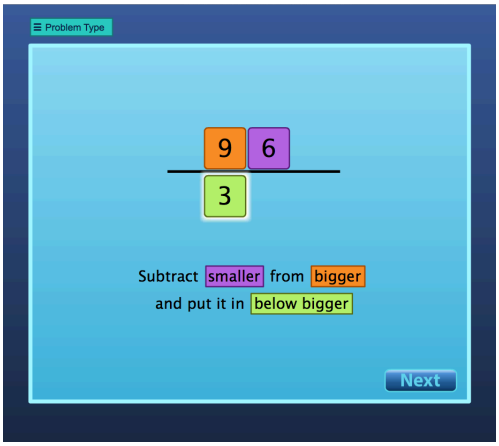


Figure 3. Screenshot of the grid mathematics application, set up to explain a greatest common factor problem with Euclid's Algorithm. The user can switch to a new procedure using the "Problem Type" menu bar.

PROTOTYPE IMPLEMENTATIONS

Our framework for automatically generating step-by-step demonstrations can theoretically be used to explain any deterministic procedural process. However, it is more appropriate for some domains than others. This approach is most effective in rich domains where many problem types can be represented through a single interface. The framework is also best suited to complex procedures with loops and branching. This represents a wide variety of problem-solving domains in K-12 education. In this section, we demonstrate how the framework applies to two distinct domains: a K-12 mathematics application and an educational game. Both applications are written in ActionScript 3, so we implemented our Thought Process Language, Compiler, and Interpreter in AS3 as well. Through these prototypes, we show that our framework can produce instructional materials for a variety of procedures and tasks without any problem-specific design.

K-12 Grid Mathematics

The domain of grid mathematics includes all problems that can be solved on a two-dimensional grid. This covers a large portion of K-12 math, including addition, subtraction, long division, prime factorization, fraction arithmetic, and even algebra. For this prototype, we implemented a general-purpose grid math application that can display a wide variety of problem types. Figure 3 shows the application interface displaying a greatest common factor problem. Users can select the desired procedure using the "Problem Type" drop-down menu. Step-by-step demonstrations for each procedure are provided for a progression of increasingly complex problems. Users step through the explanations by clicking the "next" button.

We wrote three interface hooks for this application: one that highlights an arbitrary grid cell, one that displays an explanation next to an arbitrary grid cell, and one that displays an explanation under the problem. These three simple functions can explain a large variety of procedures. For this prototype, we implemented three TPL procedures: a subtraction procedure, a multiplication procedure, and a procedure for Euclid's algorithm to find the greatest common factor of two numbers.

For each TPL procedure, our prototype can automatically generate explanations for a massive number of problems. For example, there are 55 million unique four-digit subtraction problems. Some are simple, while others require borrowing or borrowing across zero. Each type of problem is solved with a different set of steps, which is represented by a trace through our TPL algorithm. Four-digit subtraction is covered with 73 unique traces. It would require a lot of effort to author tutorials for each type of problem individually. Our approach allows us to explain all problem types automatically.

Educational Puzzle Game

Our second prototype is an educational puzzle game that is solved with a highly complex procedure. We chose this application to show how our approach scales to domains that require students to make complex decisions. *Refraction* was designed to teach fraction concepts to elementary school students. To play, the user interacts with a grid containing laser sources and target spaceships, as shown in Figure 4. The goal is to satisfy target spaceships by splitting the laser into the correct fractional amounts. The player uses pieces to change the laser direction and split the laser into fractional parts.

The most complex levels of *Refraction* cannot be solved with a deterministic procedure; heuristic search is sometimes required. For this prototype, we wrote a TPL procedure that covers the subset of *Refraction* levels that can be solved deterministically. This procedure is still much more complex than any of the grid math procedures implemented for our first prototype. It consists of 80 lines of TPL code and can solve thousands of distinct *Refraction* levels.

We wrote 27 interface hooks to explain the problem-solving process for *Refraction*. One set of functions highlights the following objects in the interface: pieces, arrays of pieces, spaceships, grid cells, grid columns or rows, spaceship fractions, laser fractions, and cardinal directions. Other functions display textual explanations that point to those same visual objects. The code we wrote in these interface hooks is very similar to the code *Refraction* developers wrote to author the game's original introductory tutorials. As a result, the authoring effort was similar to the effort required to write level-specific tutorials. However, with this approach, our functions can be used to explain any level solved by our TPL algorithm.

The textual explanations are automatically generated using explanation templates, as described in the Framework Design section. To effectively describe the complex procedure used to solve *Refraction* levels, we encoded explanations within the variable names themselves. For example, the explanation shown in Figure 4(a) was generated from the assignment of a variable named *theShipNeedsThisMuchPower*. The explanation shown in Figure 4(b) was generated from a conditional statement that calls the function $areTheseTwoValuesEqual$. By encoding textual explanations directly in the TPL algorithm's variable and function names, we can automatically generate detailed walkthroughs for any game level that can be solved by our TPL algorithm. This could improve the process of authoring tutorials and hints for educational games; previous research shows that this process can be challenging and time-consuming [19].

(a) (b) (c)

Figure 4. Screenshots of the *Refraction* prototype. To play, the student moves pieces from the bin on the right to the grid to bend and split the laser. Figure (a) shows an explanation of an assignment statement generated by our system. The text is generated from the variable name. Figure (b) shows an explanation of a conditional statement generated by our system. Figure (c) shows a summary explanation for the backtracking sub-procedure.

SCAFFOLDING MODALITIES

A central benefit of our authoring framework is that it naturally supports the creation of many different types of interactive instructional scaffolding, in addition to the step-by-step explanations described thus far. In this section, we show how our framework can be extended to produce tutorial progressions, just-in-time feedback, and fading worked examples.

Tutorial Progressions

The optimal amount of information to provide students as they work to master a new procedure varies throughout the learning process. Reigeluth and Stein argue that educators should introduce the simplest version of a new task first, and then give students progressively more complex tasks that build on the original task [22]. For novices, it is therefore desirable to minimize cognitive load by starting with simple problems that can be solved with the most straightforward procedures. For example, when introducing a simple subtraction problem like $15 - 3$, we would not mention the procedure for borrowing because it is not needed in this problem.

To produce explanations with the ideal level of detail automatically, we trace the TPL algorithm for the given problem before generating the explanations to determine which statements should be explained and which should be skipped. For each statement that branches, like a loop or a conditional, we track whether the branching behavior of this statement varies during the execution of the algorithm for the current problem. If the statement always executes the same branch, for example if the "borrowing" conditional in subtraction never evaluates to true, we skip the explanation for that statement because it is not required for the current problem.

As a student masters portions of a complex algorithm, less explanation is required. For example, an algorithm for solving linear equations will include addition and subtraction as subprocedures. However, since students studying algebra have already mastered these procedures, it is not necessary to step through them in detail. Our approach naturally supports multiple levels of explanation granularity. If the designer indicates that a sub-procedure should be summarized rather than explained in detail for a given problem, we automatically generate a summary explanation by fusing information stored in the procedure's comments. To author a sub-procedure summary, the designer writes a header comment with a high-level description of the procedure, and indicates which variables should be highlighted by adding the comment $//explain$ to the end of the assignment statement for that variable. Figure 4(c) shows the summary explanation for the backtracking procedure in *Refraction*, which is used to determine where the next piece should be placed on the grid.

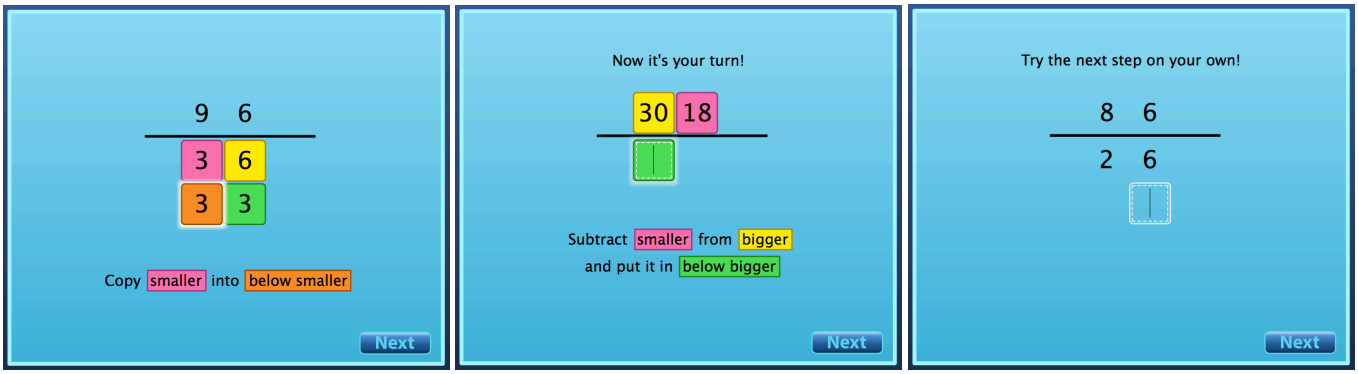
A key advantage of these techniques for producing appropriate explanations is that they build on recent work on automatically generating problem progressions for procedural tasks from an algorithmic representation of the procedure [3]. By combining our framework with these approaches, it would be possible use the TPL encoding of a procedure to automatically produce a lesson that introduces practice problems in order of increasing complexity and explains each one with an appropriate level of detail. This would remove the need to author both practice problems and their explanations by hand.

Just-in-Time Feedback

Another effective method of scaffolding learning is by providing students with feedback just as it is needed. Human tutors often give hints and suggestions when students make mistakes. Personalized hints are a central component of most cognitive tutors [26] and studies show that students perform better when these tutors provide hints [4]. Our framework naturally supports of interactive just-in-time feedback.

To show how we can generate interactive feedback directly from our step-by-step demonstrations, we created a "feedback mode" in each of our prototype applications. In this mode, the application waits for a user action and responds with a message if the action is incorrect. To author just-in-time feedback, the designer adds the comment $//action$ to each line in the TPL algorithm that modifies the user interface. For example, for the subtraction procedure in Algorithm 1, this comment would be added to lines 11, 16, 19, and 22, all of which set the value of a grid cell to a new number.

While running in "feedback mode", the interpreter executes the TPL algorithm silently, without displaying any explanations, until it reaches a line with an $//action$ comment. Then,



(a) (b) (c)

Figure 5. Screenshots of the progression for Euclid's Algorithm that gradually fades between step-by-step explanations and independent problem solving. Figure (a) shows a full demonstration, Figure (b) shows the system asking the student to perform one step with help, and Figure (c) shows independent problem solving with just-in-time feedback.

the system waits until the user performs an action. If the user action does not match the expected action, then the user has made a mistake. The designer can define how many incorrect tries the students can make before the correct steps are explained. If the student exhausts all of their attempts, the interpreter reverts the incorrect action, backtracks to the previous $//action$ comment, and displays a step-by-step demonstration of the steps the student should have made between their last correct action and their mistake.

This approach allows us to leverage the power of the interpreter to provide just-in-time feedback that directly targets student mistakes with minimal authoring effort.

Fading Worked-Out Examples

The typical method for teaching procedures begins with a full demonstration of the procedure and ends with the student solving problems without help. One limitation of this approach is that the transition between demonstrations and independent problem solving is often abrupt; students observe a teacher demonstration on the board and then solve worksheet problems without help. Some researchers have suggested that scaffolding should be used to smooth this transition [5, 23]. Renkl et al. describe a method where some steps of a procedure are demonstrated while other steps are completed by the student. By gradually reducing the number of demonstrated steps, the student eventually learns to solve the problem without help. They found that this type of gradual fading improved student performance on near-transfer tasks [23].

Initial studies that explore the effectiveness of fading worked-out examples have all used explanations and fading that were authored by hand for each problem [17]. Melis et al. have explored methods of automatically generating and adapting fading examples within the context of the ActiveMath intelligent tutoring system [17], but this approach does not generalize to other interactive learning environments or domains. Our approach naturally facilitates fading worked-out examples across multiple application domains.

We extended our framework to support fading in both of our prototype applications. To author fading for a given example, the designer indicates in the problem definition which actions in the TPL algorithm should be demonstrated and which should be performed by the student. Actions can also be fast-forwarded to focus more directly on a new part of the procedure. By defining whether each action in the TPL algorithm should be explained, performed by the user, or fast-forwarded, we can automatically create scaffolding that provides that level of fading. Figure 3 shows three different levels of fading that we created for Euclid's Algorithm in the grid math application. In Figure 5(a) the procedure is fully explained, in Figure 5(b) the student is given instructions but must complete the step on their own, and in Figure 5(c) the student solves the problem independently and receives just-in-time feedback in response to any mistakes.

The current design of our framework requires the designer to define which actions should be explained, fast-forwarded, and completed by the student for each problem. However, it could be extended further to produce faded progressions automatically. Gradual fading between worked-out examples and independent problem solving has great potential to support students as they learn new procedures, however this content is time-intensive to author and therefore difficult to study. We hope this new approach to authoring faded progressions can reduce the barrier to using and studying this technique.

USER EVALUATION

To gain a better understanding of the process of authoring scaffolding using our framework and the quality of the generated explanations, we conducted two user studies: one with educational technologists and one with fifth-grade students.

TPL Authoring User Study

The process of authoring step-by-step demonstrations with our framework consists of two parts: writing interface hook functions for the problem domain and writing a TPL algorithm for each procedure to be taught. In this study, we specifically examine the process of authoring TPL algorithms. To generate effective explanations, a TPL algorithm must abstract information into conditional blocks so that it is only explained when needed, and use variable names that describe

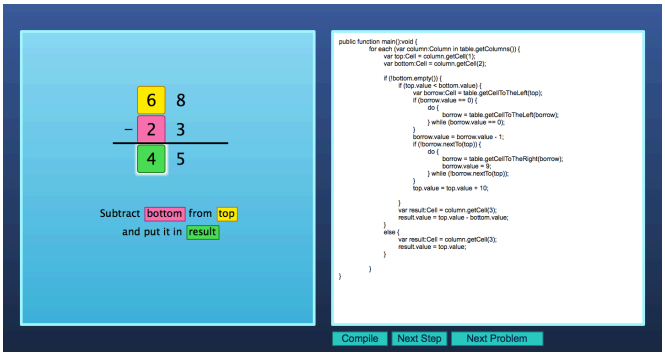


Figure 6. Screenshot of the authoring version of the grid mathematics application used in the authoring user study. Users can write Thought Process Language procedures in the panel to the right and step through the explanations they produce.

the human thought process. This requires an unusual programming style that could be challenging for authors. In contrast, authoring interface hooks requires standard programming and design skills. We were therefore most interested in evaluating whether educational technologists can learn the unique skill set required to write effective TPL algorithms.

To evaluate the process of writing TPL algorithms, we conducted a small-scale user study with three educational technologists recruited through our institution. All three participants were professional developers of educational applications with no expertise in designing instructional content. We asked participants to write a TPL procedure for subtracting one to three digit numbers that supported borrowing. We developed an authoring version of the grid mathematics application for this study, which included a code-editing panel as shown in Figure 6. To author the step-by-step demonstrations, participants wrote their TPL procedure in the code panel, clicked "Compile", and stepped through the procedure for a set of provided example problems. We gave the participants a brief overview of the system before they began the task, and provided a few resources. This included documentation for the classes that their TPL procedures could reference, descriptions of the system's interface hooks, and an example procedure for Euclid's Algorithm.

All three participants produced TPL algorithms that solved the set of example problems we provided. They spent between 2.5 and 4 hours on the task in total, a reasonable amount of time considering that their procedures could explain all three-digit subtraction problems. The explanations generated by their algorithms provided clear descriptions of the subtraction procedure, although two participants included extra steps such as counting the number of columns rather than looping over them. Most interestingly, the authors each described a slightly different thought process for solving subtraction problems, displaying our framework's ability to support multiple types of explanations. All three participants iterated on their procedures extensively during the authoring process, and were able to learn effective ways of expressing the thought process using the feedback provided by the generated explanations themselves. This shows that educational technologists are able to learn to encode the problem-solving thought process to produce high-quality step-by-step tutorials without extensive training.

While we have not formally evaluated the process of authoring interface hook functions, we found it to be straightforward for our example implementations. This process requires standard programming and design skills, and a familiarity with the target application. The effort required to author the interface hooks is highly dependent on the complexity of the application: we wrote three interface hooks for the grid math prototype and 27 for Refraction. Furthermore, interface hooks are only implemented once per problem domain, while TPL is written for each procedure to be taught.

Student User Study

The textual explanations generated by our framework are phrased slightly differently than hand-authored explanations due to the format of the underlying algorithm. We were therefore interested in evaluating whether the generated explanations can effectively teach students a new procedure. To explore this question, we collaborated with an elementary school math teacher to conduct a user study with two of his fifth-grade classes. We chose to teach Euclid's Algorithm for finding the greatest common factor of two integers because this is a procedure that is rarely taught in schools; the teacher we worked with was not familiar with the procedure.

We created two computer lessons to teach Euclid's Algorithm using our grid math prototype. One lesson displayed step-by-step demonstrations of the procedure for three example problems, and then provided a set of 18 practice problems with just-in time feedback. The second lesson presented the same 21 problems in the same order, but faded gradually between demonstrations and independent problem solving. We compared these two computer-based lessons to a traditional lesson given by the math teacher to see how the automatically-generated explanations compared to human explanations.

The Euclid's Algorithm lesson was conducted during a 50-minute math period. The first class received the traditional lesson, and the second class was split into two groups who received the two computer-based lessons. In both classes, the teacher began by giving an identical five-minute overview of factors and greatest common factors. Next, students had 10 minutes to work on a paper pre-test with three greatest common factor problems. One problem could be solved by listing the factors of each number and selecting the greatest common one, but the other two problems were designed to be difficult to solve without using Euclid's Algorithm.

After finishing the pre-test, the first class was given a traditional 25-minute lesson by their teacher. We asked him to teach Euclid's Algorithm as he would teach any new procedure, but had him introduce the same set of 21 example problems in the same order as in the computer lessons. The teacher demonstrated the procedure for two example problems on the white board, and then asked students to work on the 19 practice problems on their own while he answered questions and worked with individual students. The second class moved to computers after completing the pre-test at their

desks. Half of the students, randomly chosen, logged into a lesson with demonstrations and practice problems, while the other have logged into a lesson that faded between the demonstrations and practice problems. The students were given 15 minutes to interact with the system after logging in.

After the lessons, students went back to their desks to take the paper post-test. This test used the same three questions as the pre-test, given in a different order. We chose to use the same questions to be sure that the tests were of equal difficulty. Students were given 10 minutes to work on the post-test.

Results

We collected data from 28 students, and were given informed consent for participation by the students, their parents, their teacher, and their school principal. Of these students, 18 received the traditional lesson, 5 received the computer lesson with just-in-time feedback, and 5 received the computer lesson with fading worked-out examples. Due to a server issue we lost data from several students in the computer class, hence the smaller number of participants in these conditions.

We used non-parametric statistics in our analysis because the students' scores on the pre- and post-tests were non-normally distributed. We found that overall, scores improved between the pre- and post-test. We used a Wilcoxon rank sums test to measure the effect of *test* (either pre or post) on scores, showing that students performed significantly better on the post-test (Z=-3.80, $p<0.0001$), with a median of 2.5 questions correct out of three, compared to 0 correct on the pre-test. We also calculated a learning gain for each student by taking the difference between their pre- and post-test scores, and found that the median learning gain was 1 point.

Next, we used a Kruskal-Wallis test to determine whether *condition* had any impact on students' scores. There was no significant difference in pre-test scores (χ^2=3.11, *n.s.*). Students in the traditional lesson condition got a median of 1 problem correct, compared to 0 for the two computer lesson conditions. There was also no significant difference in post-test scores (χ^2=3.11, *n.s.*); students in the traditional lesson condition got a median of 3 problems correct, compared to 2 problems correct for students in the fading computer lesson, and 1 problem correct for students in the just-in-time feedback computer lesson. Finally, *condition* had no significant impact on learning gains (χ^2=0.01 *n.s.*). Students in all three conditions had a median learning gain of 1 point.

Our results show that students who received the two computer lessons had learning gains roughly equivalent to those who received the traditional lesson. Only two students did not use Euclid's Algorithm to solve the post-test problems, one in the traditional lesson condition and one in the just-in-time feedback condition. This shows that the generated explanations were able to effectively convey a new procedure to students, even over a very short period of time. Students had less time-on-task and no help from their teacher with the computer lessons, but still performed equally well on the post-test.

While these results are promising, our sample size is small and we only looked at one TPL algorithm in this study. In future work, it would be valuable to conduct a more thorough experiment that focuses on the student experience. This would help us gain a better understanding of the strengths and weaknesses of our automatically generated explanations.

CONCLUSION

In this work, we present a new framework for automatically generating interactive scaffolding to teach the entire space of problems that can be solved with a given procedure. We encode the problem-solving thought process as an algorithm and use this model to generate explanations. This approach has a number of advantages. Given a small set of mappings between programming language constructs and visual objects in the user interface, we can produce tutorials for any Thought Process Language procedure that can be applied in the application. Furthermore, this framework naturally facilitates the authoring of problem progressions that introduce information gradually, just-in-time feedback, and scaffolding that fades between demonstrations and independent problem solving.

We demonstrate the generality of our approach by showing how the framework applies to two distinct problem-solving domains: a K-12 math application and an educational game. For each prototype, we authored interface hooks and TPL algorithms that produce in-depth explanations for problem-solving procedures in that domain. We also conduced an authoring user study to determine whether educational technologists can effectively encode a problem-solving procedure in our Thought Process Language. All three participants produced high-quality TPL procedures to explain subtraction in a reasonable amount of time. Finally, we conducted a user study with fifth grade students to determine whether the explanations generated by our framework can effectively teach a new procedure. This study showed that students who received computer lessons had similar learning gains to students who received a traditional lesson from their math teacher.

Our new approach for authoring interactive instructional scaffolding is designed to maximize the content that educational technologists with programming expertise can create. The authoring effort required to write interface hooks and TPL algorithms will not be not justified in all domains. However, our framework has incredible potential in domains where we would like to generate scaffolding for a wide variety of example problems. Perhaps most importantly, this approach naturally supports the creation of a variety of different types of instructional scaffolding, including problem progressions, just-in-time feedback, and fading worked-out examples. Since these types of scaffolding are costly to produce, many important questions about the relative effectiveness of different scaffolding methods remain unanswered. We hope that this framework will be used to improve tutorial authoring and also expand our understanding of effective scaffolding design.

ACKNOWLEDGMENTS

We would like to thank our study participants, the creators of *Refraction*, and Craig Connor for his work on the grid mathematics prototype. This work was supported by the Office of Naval Research grant N00014-12-C-0158, the Bill and Melinda Gates Foundation grant OPP1031488, the Hewlett Foundation grant 2012-8161, Adobe, and Microsoft.

REFERENCES

1. Aleven, V., McLaren, B. M., Sewall, J., and Koedinger, K. R. The cognitive tutor authoring tools (CTAT): preliminary evaluation of efficiency gains. In *International Conference on Intelligent Tutoring Systems*, ITS'06 (2006), 61–70.
2. Aleven, V., Mclaren, B. M., Sewall, J., and Koedinger, K. R. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *Int. J. Artif. Intell. Ed. 19*, 2 (Apr. 2009), 105–154.
3. Andersen, E., Gulwani, S., and Popović, Z. A trace-based framework for analyzing and synthesizing educational progressions. In *CHI* (2013), 773–782.
4. Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R. Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences 4*, 2 (1995), 167–207.
5. Atkinson, R. K., Renkl, A., and Merrill, M. M. Transitioning from Studying Examples to Solving Problems: Effects of Self-Explanation Prompts and Fading Worked-Out Steps. *Journal of Educational Psychology 95*, 4 (2003), 774–83.
6. Bergman, L., Castelli, V., Lau, T., and Oblinger, D. Docwizards: A system for authoring follow-me documentation wizards. In *UIST* (2005), 191–200.
7. Blessing, S. B., Gilbert, S. B., Ourada, S., and Ritter, S. Authoring model-tracing cognitive tutors. *International Journal of Artificial Intelligence in Education 19*, 2 (2009), 189 – 210.
8. Brawner, K., Holden, H., Goldberg, B., and Sottilare, R. Recommendations for modern tools to author tutoring systems. In *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)* (2012).
9. Chi, P.-Y., Ahn, S., Ren, A., Dontcheva, M., Li, W., and Hartmann, B. MixT: automatic generation of step-by-step mixed media tutorials. In *UIST* (2012), 93–102.
10. Corbett, A., Koedinger, K. R., and Anderson, J. R. Intelligent tutoring systems. In *Handbook of Human-Computer Interaction, Second Edition*, M. Helander, T. K. Landauer, and P. Prahu, Eds. Elsevier Science, Amsterdam, 1997, 849–874.
11. Glazewski, K. D., and Ertmer, P. A. Scaffolding disciplined inquiry in problem-based learning environments. *International Journal of Learning 12*, 6 (2005), 297–306.
12. Grabler, F., Agrawala, M., Li, W., Dontcheva, M., and Igarashi, T. Generating photo manipulation tutorials by demonstration. In *ACM SIGGRAPH* (2009), 66:1–66:9.
13. Harms, K. J., Cosgrove, D., Gray, S., and Kelleher, C. Automatically generating tutorials to enable middle school children to learn programming independently. In *International Conference on Interaction Design and Children*, IDC '13 (2013), 11–19.
14. Kelleher, C., and Pausch, R. Stencils-based tutorials: Design and evaluation. In *CHI* (2005), 541–550.
15. Koedinger, K. R., Anderson, J. R., Hadley, W. H., and Mark, M. A. Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education 8* (1997), 30–43.
16. Li, W., Zhang, Y., and Fitzmaurice, G. Tutorialplan: Automated tutorial generation from cad drawings. In *IJCAI*, F. Rossi, Ed., IJCAI/AAAI (2013), 2020–2027.
17. Melis, E., Goguadze, G., and Saarl, U. D. Towards adaptive generation of faded examples. In *International Conference on Intelligent Tutoring Systems , LNCS*, Springer- Verlag (2004), 762–771.
18. Olsen, J., Belenky, D., Aleven, V., Rummel, N., Sewall, J., and Ringenberg, M. Authoring tools for collaborative intelligent tutoring system environments. In *Intelligent Tutoring Systems* (2014), 523–528.
19. O'Rourke, E., Ballweber, C., and Popović, Z. Hint systems may negatively impact performance in educational games. In *Proceedings of Learning @ Scale*, L@S '14 (2014), 51–60.
20. Pongnumkul, S., Dontcheva, M., Li, W., Wang, J., Bourdev, L., Avidan, S., and Cohen, M. F. Pause-and-play: automatically linking screencast video tutorials with applications. In *UIST* (2011), 135–144.
21. Ramesh, V., Hsu, C., Agrawala, M., and Hartmann, B. Showmehow: Translating user interface instructions between applications. In *UIST* (2011), 127–134.
22. Reigeluth, C. M., and Stein, F. S. The elaboration theory of instruction. In *Instructional Design Theories and Models: An Overview of their Current States*, Lawrence Erlbaum (Hillsdale, NJ, 1983).
23. Renkl, A., Atkinson, R. K., Maier, U. H., and Staley, R. From example study to problem solving: Smooth transitions help learning. *Journal of Experimental Education 70*, 4 (2002), 293–315.
24. Sottilare, R. A. Considerations in the development of an ontology for a generalized intelligent framework for tutoring. In *International Defense & Homeland Security Simulation Workshop in Proceedings of the I3M Conference* (2012), 19–25.
25. Stamper, J., Barnes, T., Lehmann, L., and Croy, M. The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track* (2008), 71–78.
26. VanLehn, K. The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education 16* (2006), 227–265.
27. Wood, D., Bruner, J., and Ross, G. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry 17* (1976), 89–100.